

**CS 441/541**  
**Artificial Intelligence**  
**Fall, 2008**

**Homework 5:**  
**Neural Networks and Face Recognition**

Due Wednesday, Nov. 5.

This set of exercises is adapted from Tom Mitchell's neural network homework assignment given at

<http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html>

The code and code documentation that goes with this assignment is also taken from that page.

This code will definitely compile and run on the PSU CECS Linux Lab machines. I can't vouch for it on any other platform. (However, see the file included in the tarball called "WindowsReadme.txt", which explains how to get the code to run on Windows.)

### **Introduction**

This assignment gives you an opportunity to apply neural network learning to the problem of face recognition, as was discussed in class.

You will use a neural network system written in C by Jeff Shufelt of Carnegie Mellon University.

The code and all training and testing data is in gzipped tarball form and can be downloaded from the class web site:

<http://www.cs.pdx.edu/~mm/ArtificialIntelligenceFall2008/HW5.tgz>

The documentation for the code is given at the end of this document.

## Contents of code tarball

The contents of the tarball you will download contains

- The .c and .h files that implement the neural network and associated routines;
- The images of faces, contained in a subdirectory called “faces”;
- Several “.list” files that list subsets of the faces to be used for training and testing data.

The C code creates and trains a three-layer fully-connected feedforward neural network, using the backpropagation algorithm to tune its weights. It also contains an image package for accessing the face images, as well as the top-level program for training and testing. To help explore what the nets actually learn, you’ll also find a utility program for visualizing hidden-unit weights as images.

The “faces” subdirectory contains 20 subdirectories, each containing images corresponding to a particular person. Each of these images has four features associated with it: the name of the person, the direction they are facing, their facial expression, and whether or not they are wearing sunglasses or are “open” (not wearing sunglasses). The feature values for each image are given in the file name. For example, in the directory “faces/bpm/” two of the files are:

```
bpm_up_happy_sunglasses_4.pgm  
bpm_left_angry_open_4.pgm
```

The first of these indicates that the image is of person “bpm” looking up with a happy expression and wearing sunglasses. The second indicates that the image is of person “bpm” looking left with an angry expression and not wearing sunglasses.

You can view any of the image files on Linux machines using the `display` command.

## How to run the code

1. Create a new subdirectory for this assignment.
2. Download the tarball into this subdirectory and untar it.
3. Type “make”. This will compile the code. (Ignore compiler warnings.)
4. The code is currently set up to create a neural network with 940 input units, 3 hidden units, and one output unit, as described in class and in the reading on face recognition. To run the code, type

```
facetrain -n sunglasses.net -s 1000 -t all_train.list -1 all_test1.list
```

This will create a neural network as described above, using random number seed “1000”, and train it for the default of 100 epochs on the training data listed in “all\_train.list”. The resulting network will be stored in the file “sunglasses.net”.

The output will look like:

```
Loading 'faces/kawamura/kawamura_straight_happy_open_4.pgm'...done
Loading 'faces/phoebe/phoebe_up_sad_open_4.pgm'...done
    < many more such 'loading' lines >
Random number generator seed: 1000
277 images in training set
139 images in test1 set
0 images in test2 set
Creating new network 'sunglasses.net'
Training underway (going to 100 epochs)
Will save network every 100 epochs
0 0.0 50.9025 0.0972029 46.7626 0.103391 0.0 0.0
1 29.4377 50.9025 0.07842 47.482 0.0805035 0.0 0.0
2 30.6895 59.2058 0.0737351 53.2374 0.0763508 0.0 0.0
3 31.2238 67.87 0.0665212 63.3094 0.0704276 0.0 0.0
    ...
100 2.10745 99.639 0.00257808 85.6115 0.0348871 0.0 0.0
Saving 960x4x1 network to 'sunglasses.net'

Saving 960x4x1 network to 'sunglasses.net'
```

## Running facetrain

As illustrated above, `facetrain` has several options which can be specified on the command line. This section briefly describes how each option works. A very short summary of this information can be obtained by running `facetrain` with no arguments.

- n `<network file>` - this option either loads an existing network file, or creates a new one with the given name. At the end of training, the neural network will be saved to this file.
- e `<number of epochs>` - this option specifies the number of training epochs which will be run. If this option is not specified, the default is 100.
- T - for test-only mode (no training). Performance will be reported on each of the three datasets specified, and those images misclassified will be listed, along with the corresponding output unit levels.
- s `<seed>` - an integer which will be used as the seed for the random number generator. The default seed is 102194 (guess what day it was when I wrote this document). This allows you to reproduce experiments if necessary, by generating the same sequence of random numbers. It also allows you to try a different set of random numbers by changing the seed.
- S `<number of epochs between saves>` - this option specifies the number of epochs between saves. The default is 100, which means that if you train for 100 epochs (also the default), the network is only saved when training is completed.
- t `<training image list>` - this option specifies a text file which contains a list of image pathnames, one per line, that will be used for training. If this option is not specified, it is assumed that no training will take place (*epochs* = 0), and the network will simply be run on the test sets. In this case, the statistics for the training set will all be zeros.
- 1 `<test set 1 list>` - this option specifies a text file which contains a list of image pathnames, one per line, that will be used as a test set. If this option is not specified, the statistics for test set 1 will all be zeros.

-2 <test set 2 list> - same as above, but for test set 2. The idea behind having two test sets is that one can be used as part of the train/test paradigm, in which training is stopped when performance on the test set begins to degrade. The other can then be used as a “real” test of the resulting network.

## Interpreting the output of facetrain

When you run `facetrain`, it will first read in all the data files and print a bunch of lines regarding these operations. Once all the data is loaded, it will begin training. At this point, the network’s training and test set performance is outlined in one line per epoch. For each epoch, the following performance measures are output:

<epoch> <delta> <trainperf> <trainerr> <t1perf> <t1err> <t2perf>  
<t2err>

These values have the following meanings:

`epoch` is the number of the epoch just completed; it follows that a value of 0 means that no training has yet been performed.

`delta` is the sum of all  $\delta$  values on the hidden and output units as computed during backprop, over all training examples for that epoch.

`trainperf` is the percentage of examples in the training set which were correctly classified.

`trainerr` is the average, over all training examples, of the error function  $\frac{1}{2} \sum_i (t_i - o_i)^2$ , where  $t_i$  is the target value for output unit  $i$  and  $o_i$  is the actual output value for that unit.

`t1perf` is the percentage of examples in test set 1 which were correctly classified.

`t1err` is the average, over all examples in test set 1, of the error function described above.

`t2perf` is the percentage of examples in test set 2 which were correctly classified.

`t2err` is the average, over all examples in test set 2, of the error function described above.

For this assignment, you will only need to be concerned with the epoch numbers, the accuracy on the training set and the accuracy on test set 1.

## Your assignment

Your assignment is to perform several experiments with `facetrain` and do some analysis of the results.

## Experiments

For each experiment listed below, you need to do 10 runs of `facetrain` (using different random number seeds) for 100 epochs each and report (1) the average and standard deviation of the network's accuracy on the training data (2) the average and standard deviation of the accuracy on test set 1.

### 1. Running the code:

Run the network using the default parameters given above: 940 input units, 3 hidden units, and one output unit. To do this, you simply need to type:

```
facetrain -n sunglasses1.net -s 1000 -t all_train.list -1 all_test1.list
```

Repeat this for a total of 10 runs. Each time you repeat a run, change the name of the output network and the random number seed. An easy way to do this is to use the sequence of names “`sunglasses1.net`”, “`sunglasses2.net`”, ..., “`sunglasses10.net`”. and random number seeds 1001, 1002, ... 1009.

### 2. Varying the number of hidden units

a. Change the number of hidden units to 2. This can be done by editing the file `facetrain.c`, and search for “`EDIT`”. This shows you which line to edit to change the number of hidden units. Recompile the code with `make` and run with the following command:

```
facetrain -n sunglasses1-hidden2.net -s 2000 -t all_train.list
-1 all_test1.list
```

Note that I have changed the name of the network output file to “sunglasses1-hidden2.net” and the random number seed to 2000. Do a total of 10 runs of this with different network names and random number seeds.

Remember to recompile the code each time you make a change.

b. Repeat experiment 2a, but now with 20 hidden units. Does changing the number of hidden units affect your results on training or generalization performance? Why or why not?

### 3. Varying the number of training examples

Reset `facetrain.c` so that it creates a network with 3 hidden units.

a. Create a new `.list` file with one-half of the training examples of `all_train.list` (i.e., 139). Run `facetrain` on this new training set as in experiment 2, reporting averages and standard deviations as before.

b. Repeat experiment 3a but now with one-tenth of the training examples of `all_train.list` (28). Choose any examples, but make sure about half have sunglasses and half don't.

Briefly discuss how your results change as you change the number of training examples, and why.

### 4. Using a different target function

Now change the code in `imagenet.c` to create a new target function of your choice, based on one of the other attributes. (For example, you could make the target function “classify whether or not a face is ‘happy’”.) Repeat experiment 1 with this new target function. Is your new target function easier, harder, or the same difficulty as the sunglasses function? Speculate briefly on why you got the results you obtain.

### 5. Analyzing the weights

For the network in experiment 4, try taking a look at how back-propagation tuned the weights of the hidden units with respect to each pixel. First type `make hidtopgm` to compile the utility on your system. Then, to visualize the weights of hidden unit  $n$ , type:

```
hidtopgm <networkname>.net image-filename 32 30 n
```

where  $n$  ranges from 1 to *number of hidden units*. Remember that hidden unit 0 is the bias unit.

(You can convert from pgm to other image formats using the “convert” command.)

Invoking `display` on the image *image-filename* should then display the range of weights, with the lowest weights mapped to pixel values of zero, and the highest mapped to 255. If the images just look like noise, try retraining using `facetrain_init0` (compile with `make facetrain_init0`), which initializes the hidden unit weights of a new network to zero, rather than random values.

Do the hidden units seem to weight particular regions of the image greater than others? Do particular hidden units seem to be tuned to different features of some sort? Discuss your results.

Now look at the weights on the links to the output unit. To do this, use the program `outtopgm`. Type `make outtopgm`, and then

```
outtopgm <networkname>.net image-filename 4 1 n
```

where  $n$  ranges from 1 to *number of output unit*. (4 in the above command refers the number of weights for connections from three hidden units to the output unit, plus the connection from the bias unit.) For a network with one output unit, use

```
outtopgm <networkname>.net image-filename 4 1 1
```

Discuss what information you can glean from looking at these weights together with the input-to-hidden layer weights you discussed above. (See the reserved reading on neural networks and this code for an example of such an analysis.)

## What to turn in

Turn in a clearly written and formatted writeup with a summary of your results and discussions asked for above. Include the pictures you produce in the final experiment.

## Code documentation

This documentation is taken from

<http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html>

Most of this documentation contains details you don't need to know for the assignment, but is included for the sake of completeness.

### The neural network package

As mentioned earlier, this package implements three-layer fully-connected feedforward neural networks, using a backpropagation weight tuning method. We begin with a brief description of the data structure, a BPNN (BackPropNeuralNet).

All unit values and weight values are stored as `doubles` in a BPNN.

Given a BPNN `*net`, you can get the number of input, hidden, and output units with `net->input_n`, `net->hidden_n`, and `net->output_n`, respectively.

Units are all indexed from 1 to  $n$ , where  $n$  is the number of units in the layer. To get the value of the  $k$ th unit in the input, hidden, or output layer, use `net->input_units[k]`, `net->hidden_units[k]`, or `net->output_units[k]`, respectively.

The target vector is assumed to have the same number of values as the number of units in the output layer, and it can be accessed via `net->target`. The  $k$ th target value can be accessed by `net->target[k]`.

To get the value of the weight connecting the  $i$ th input unit to the  $j$ th hidden unit, use `net->input_weights[i][j]`. To get the value of the weight connecting the  $j$ th hidden unit to the  $k$ th output unit, use `net->hidden_weights[j][k]`.

The routines are as follows:

```
void bpnn_initialize(seed)
    int seed;
```

This routine initializes the neural network package. It should be called before any other routines in the package are used. Currently, its sole

purpose in life is to initialize the random number generator with the input seed.

```
BPNN *bpnn_create(n_in, n_hidden, n_out)
    int n_in, n_hidden, n_out;
```

Creates a new network with `n_in` input units, `n_hidden` hidden units, and `n_output` output units. All weights in the network are randomly initialized to values in the range  $[-1.0, 1.0]$ . Returns a pointer to the network structure. Returns `NULL` if the routine fails.

```
void bpnn_free(net)
    BPNN *net;
```

Takes a pointer to a network, and frees all memory associated with the network.

```
void bpnn_train(net, learning_rate, momentum, erro, errh)
    BPNN *net;
    double learning_rate, momentum;
    double *erro, *errh;
```

Given a pointer to a network, runs one pass of the backpropagation algorithm. Assumes that the input units and target layer have been properly set up. `learning_rate` and `momentum` are assumed to be values between 0.0 and 1.0. `erro` and `errh` are pointers to doubles, which are set to the sum of the  $\delta$  error values on the output units and hidden units, respectively.

```
void bpnn_feedforward(net)
    BPNN *net;
```

Given a pointer to a network, runs the network on its current input values.

```
BPNN *bpnn_read(filename)
    char *filename;
```

Given a filename, allocates space for a network, initializes it with the weights stored in the network file, and returns a pointer to this new BPNN. Returns `NULL` on failure.

```
void bpnn_save(net, filename)
```

```
    BPNN *net;
```

```
    char *filename;
```

Given a pointer to a network and a filename, saves the network to that file.

## The image package

The image package provides a set of routines for manipulating PGM images. An image is a rectangular grid of pixels; each pixel has an integer value ranging from 0 to 255. Images are indexed by rows and columns; row 0 is the top row of the image, column 0 is the left column of the image.

```
IMAGE *img_open(filename)
```

```
    char *filename;
```

Opens the image given by `filename`, loads it into a new `IMAGE` data structure, and returns a pointer to this new structure. Returns `NULL` on failure.

```
IMAGE *img_creat(filename, nrows, ncols)
```

```
    char *filename;
```

```
    int nrows, ncols;
```

Creates an image in memory, with the given filename, of dimensions `nrows`  $\times$  `ncols`, and returns a pointer to this image. All pixels are initialized to 0. Returns `NULL` on failure.

```
int ROWS(img)
```

```
    IMAGE *img;
```

Given a pointer to an image, returns the number of rows the image has.

```
int COLS(img)
```

```
    IMAGE *img;
```

Given a pointer to an image, returns the number of columns the image has.

```
char *NAME(img)
    IMAGE *img;
```

Given a pointer to an image, returns a pointer to its base filename (i.e., if the full filename is `/usr/joe/stuff/foo.pgm`, a pointer to the string `foo.pgm` will be returned).

```
int img_getpixel(img, row, col)
    IMAGE *img;
    int row, col;
```

Given a pointer to an image and row/column coordinates, this routine returns the value of the pixel at those coordinates in the image.

```
void img_setpixel(img, row, col, value)
    IMAGE *img;
    int row, col, value;
```

Given a pointer to an image and row/column coordinates, and an integer value assumed to be in the range `[0, 255]`, this routine sets the pixel at those coordinates in the image to the given value.

```
int img_write(img, filename)
    IMAGE *img;
    char *filename;
```

Given a pointer to an image and a filename, writes the image to disk with the given filename. Returns 1 on success, 0 on failure.

```
void img_free(img)
    IMAGE *img;
```

Given a pointer to an image, deallocates all of its associated memory.

```
IMAGELIST *imgl_alloc()
```

Returns a pointer to a new `IMAGELIST` structure, which is really just an array of pointers to images. Given an `IMAGELIST *il`, `il->n` is the number of images in the list. `il->list[k]` is the pointer to the `k`th image in the list.

```
void imgl_add(il, img)
    IMAGELIST *il;
    IMAGE *img;
```

Given a pointer to an imagelist and a pointer to an image, adds the image at the end of the imagelist.

```
void imgl_free(il)
    IMAGELIST *il;
```

Given a pointer to an imagelist, frees it. Note that this does not free any images to which the list points.

```
void imgl_load_images_from_textfile(il, filename)
    IMAGELIST *il;
    char *filename;
```

Takes a pointer to an imagelist and a filename. `filename` is assumed to specify a file which is a list of pathnames of images, one to a line. Each image file in this list is loaded into memory and added to the imagelist `il`.

## hidtopgm

`hidtopgm` takes the following fixed set of arguments:

```
hidtopgm net-file image-file x y n
```

*net-file* is the file containing the network in which the hidden unit weights are to be found.

*image-file* is the file to which the derived image will be output.

*x* and *y* are the dimensions in pixels of the image on which the network was trained.

*n* is the number of the target hidden unit. *n* may range from 1 to the total number of hidden units in the network.

## outtopgm

`outtopgm` takes the following fixed set of arguments:

```
outtopgm net-file image-file x y n
```

This is the same as hidtopgm, for output units instead of input units. Be sure you specify  $x$  to be 1 plus the number of hidden units, so that you get to see the weight  $w_0$  as well as weights associated with the hidden units. For example, to see the weights for output number 2 of a network containing 3 hidden units, do this:

```
outtopgm pose.net pose-out2.pgm 4 1 2
```

*net-file* is the file containing the network in which the hidden unit weights are to be found.

*image-file* is the file to which the derived image will be output.

$x$  and  $y$  are the dimensions of the hidden units, where  $x$  is always 1 + the number of hidden units specified for the network, and  $y$  is always 1.

$n$  is the number of the target output unit.  $n$  may range from 1 to the total number of output units for the network.